02180: Introduction to Artificial Intelligence

Lecture 2: Uninformed Search

Nina Gierasimczuk

Problem Solving

Problem solving is an essential part of intelligence.

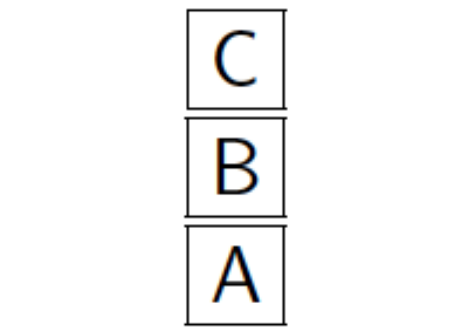
Wolf, goat and cabbage problem

Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage. If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.

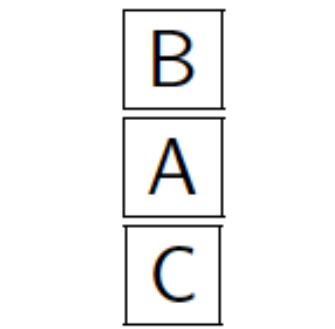
The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?

Search problem informally

Given an **initial state**, a **goal state** (or set of goal states) and a set of **available actions**, find a sequence of actions that take you from the initial state to the goal state.



Initial state s_0



Goal state

Search problems formally

Formally, a **search problem** has 5 components:

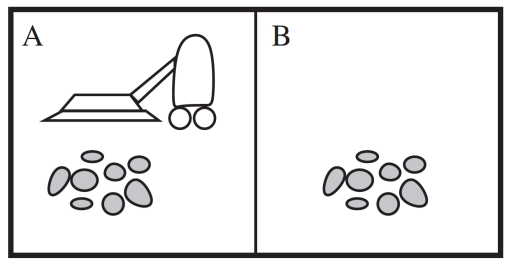
- s_0: Initial state
- $\textsc{Actions}(s)$: Returns the set of actions **applicable** in state s. (E.g. the action of opening a door might only be applicable if the door is unlocked.)
- $\textsc{Results}(s, a)$: Returns the state s' reached from s by executing action a.
- $\textsc{Goal-Test}(s)$: Returns `true` if s is goal state, otherwise `false`.
- $\textsc{Step-Cost}(s, a)$: The cost of executing action a in s. Most often we will assume $\textsc{Step-Cost}(s, a) = 1$ for all s and a.

A state g is called a **goal state** if $\textsc{Goal-Test}(g) = \texttt{true}$.

A **solution** to a search problem is a **sequence of actions** (a **path**) from s_0 to a goal state. It is **optimal** if it has minimum sum of step costs.

A Toy Problem: Vacuum World

Vacuum World consists of two locations, each of which may or may not contain dirt and the vacuum is in one of the locations.



A Toy Problem: Vacuum World

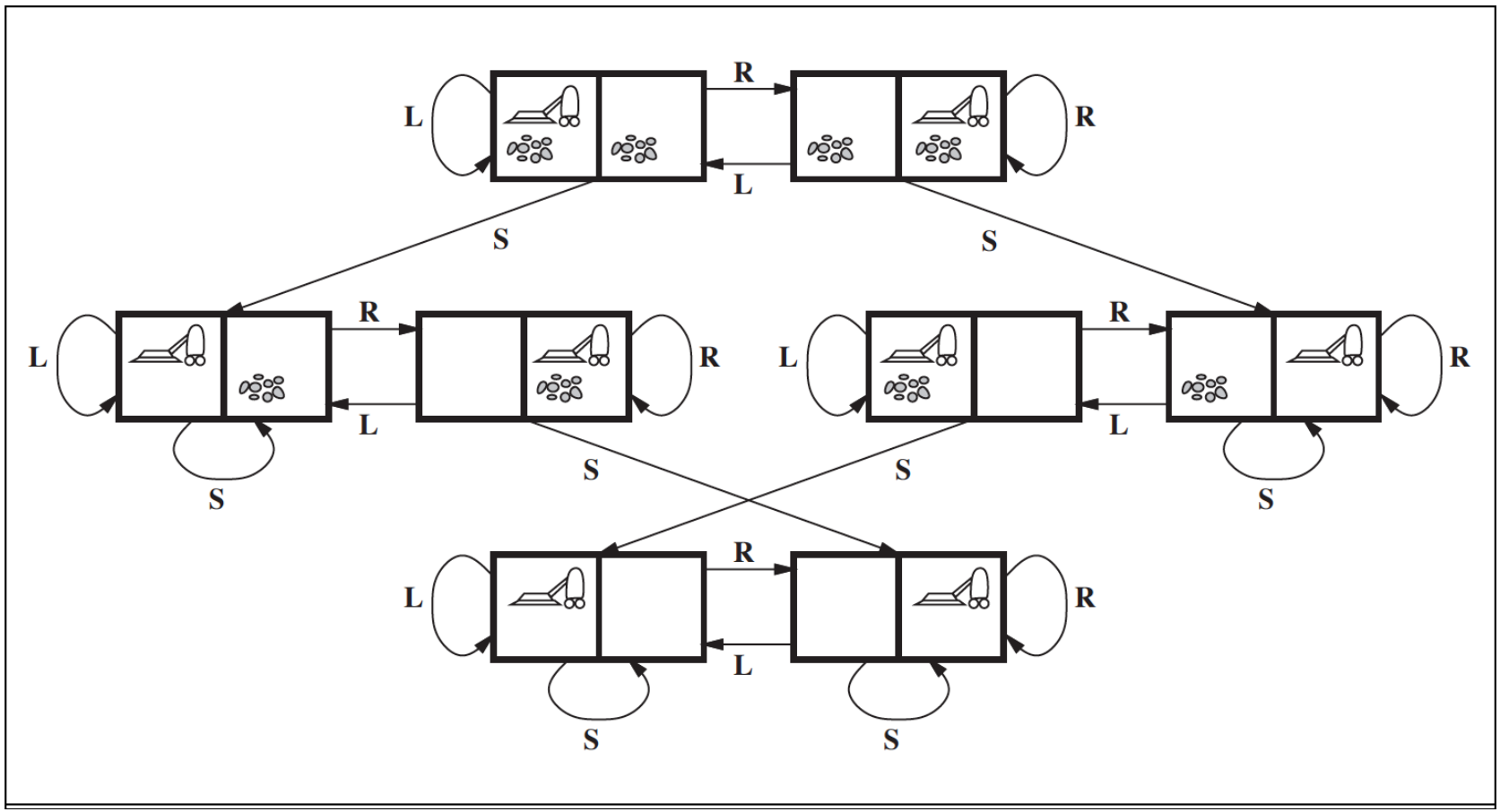
States space consists of each possible configuration (2×2^2 possible states).

- s_0: Initial state
- Actions(s): for each state three possible actions: L, R, S.
- Results(s, a): actions have their expected results.
- Goal-Test(s): *are all squares clean?*
- Step-Cost(s, a): each step costs 1.

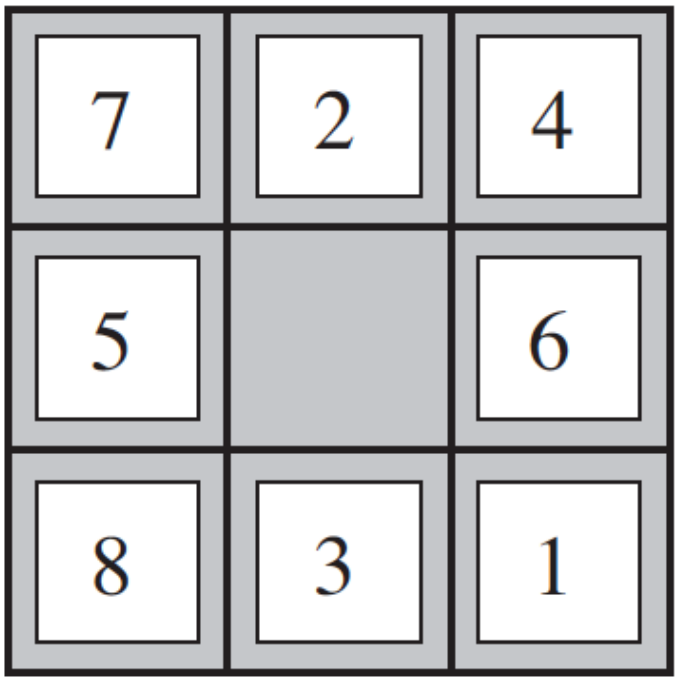
A Toy Problem: Vacuum World

Transition Model

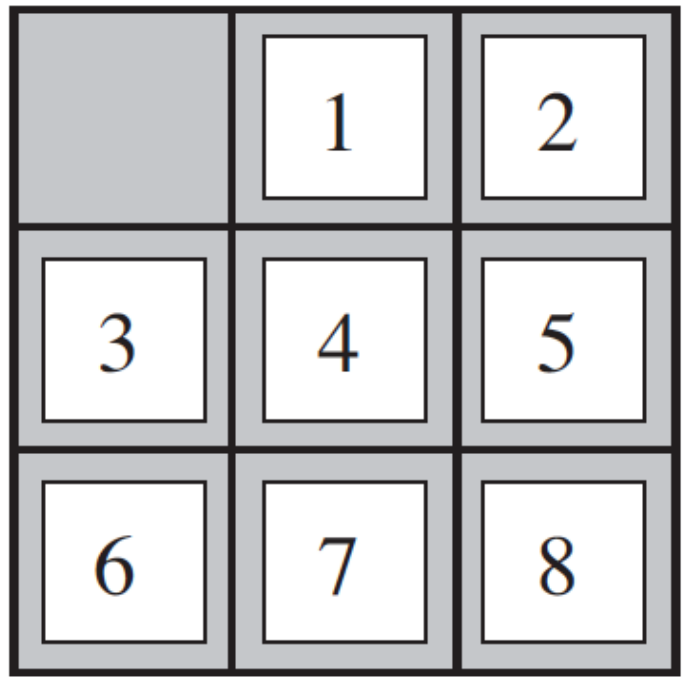
The transitions in the Vacuum World can be represented as a graph.



Another Toy Problem: 8-puzzle

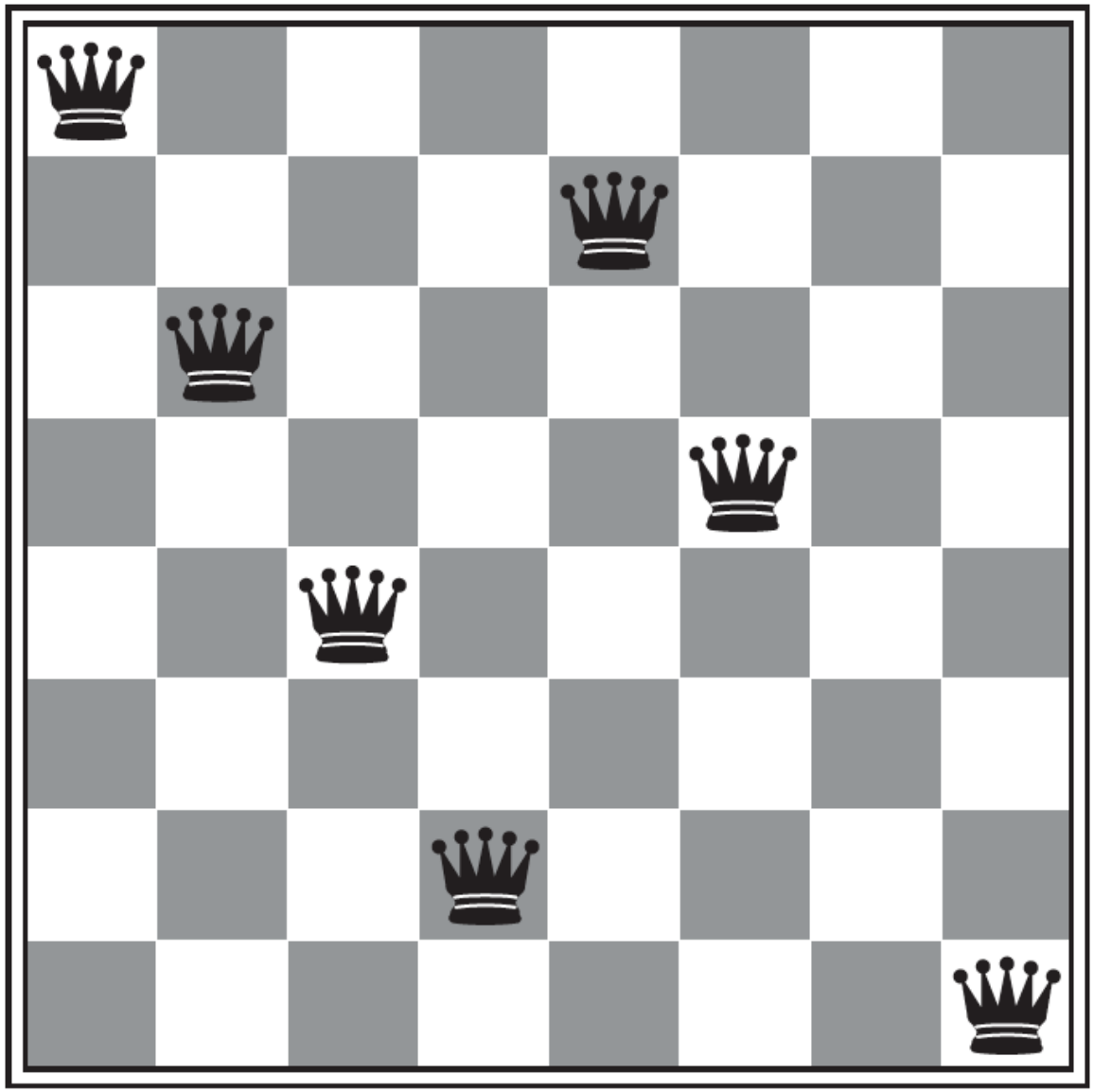


Start State



Goal State

Yet Another Toy Problem: 8-queens

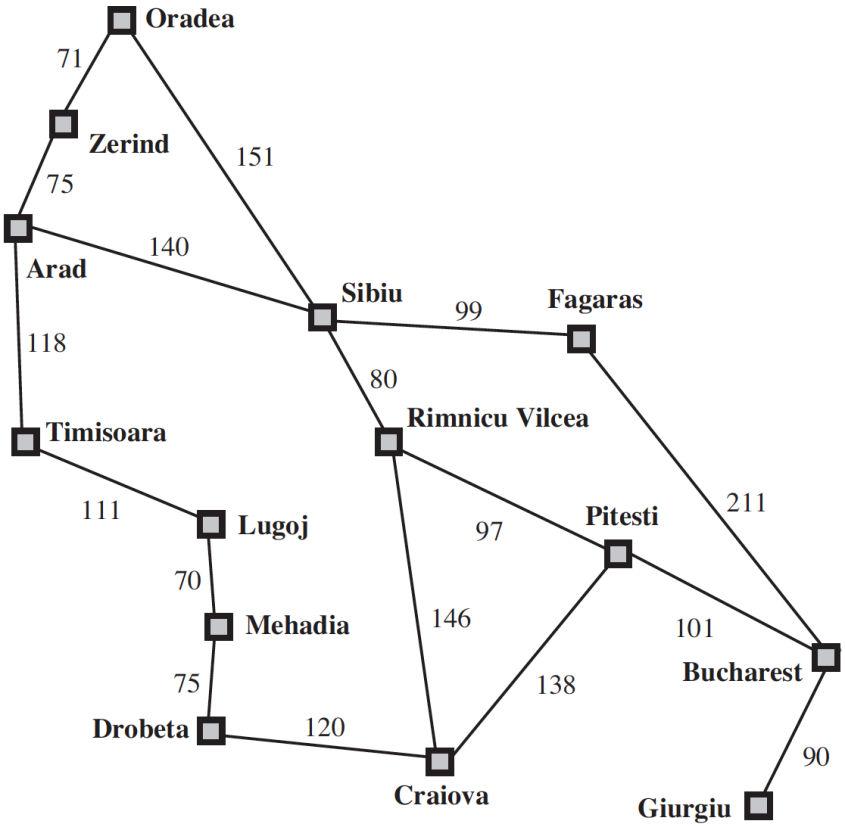


Levels of Abstraction

Levels of Abstraction

A Real World Problem

Route-finding



A Real World Problem

Route-finding

States include a location and the current time, etc.

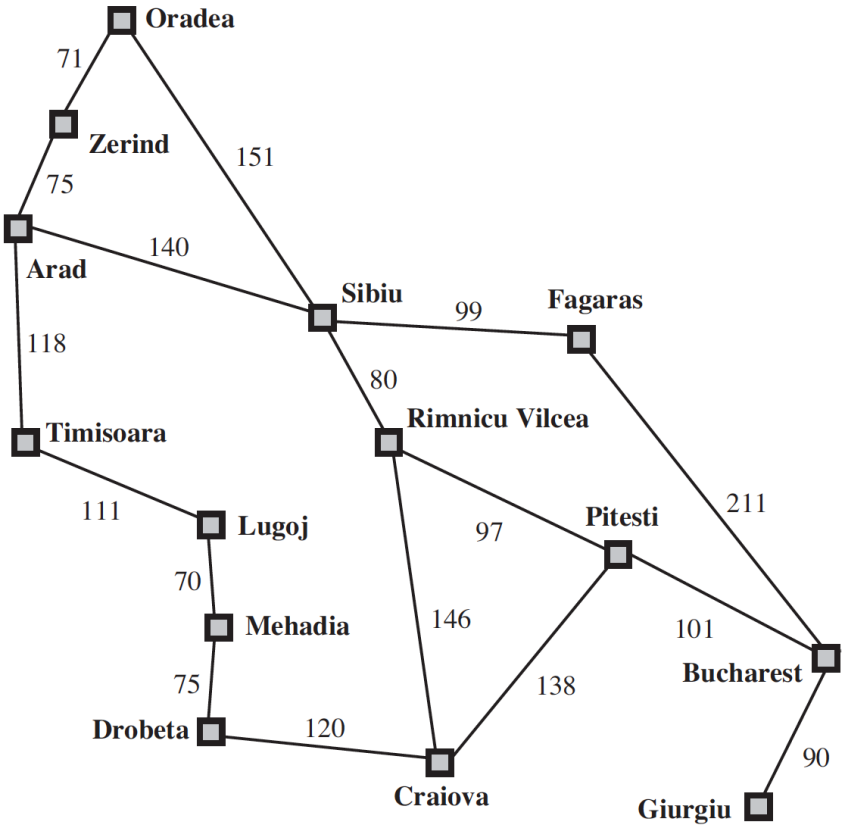
- s_0: specified by the user's query.
- Actions(s): take any flight from the current location, in any class, etc.
- Results(s, a): the state resulting from taking a flight.

  flight destination as current location and flight arrival time as current time.
- Goal-Test(s): *Are we at the final destination specified by the user?*
- Step-Cost(s, a): monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, etc.

Another Real World Problem

Touring Problem

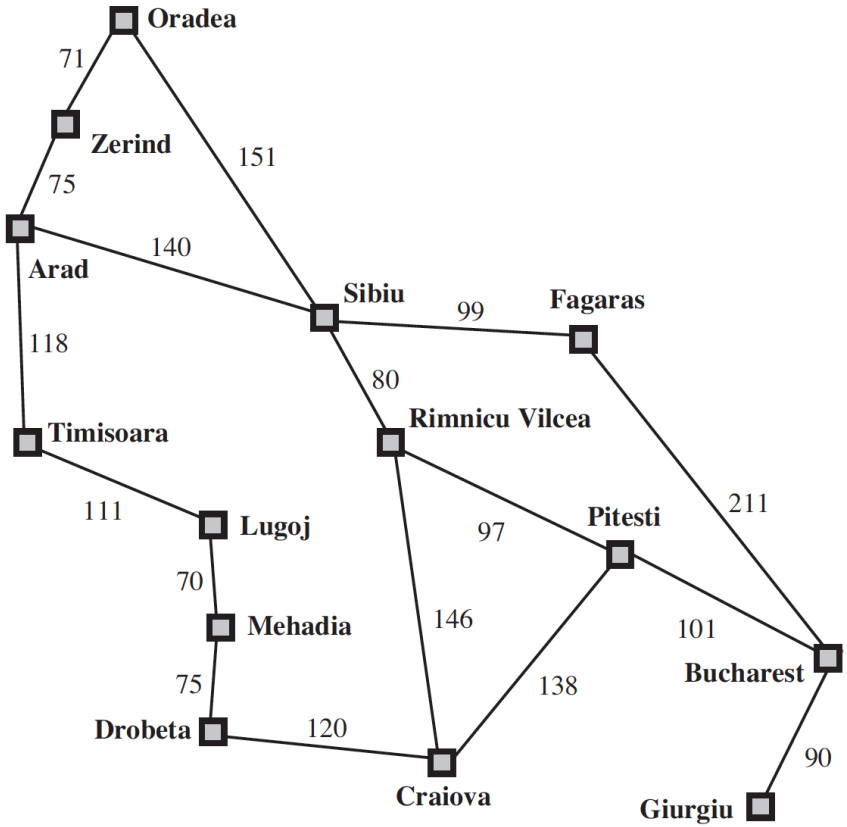
Visit every city in the figure **at least once**, starting and ending in Bucharest.



Yet Another Real World Problem

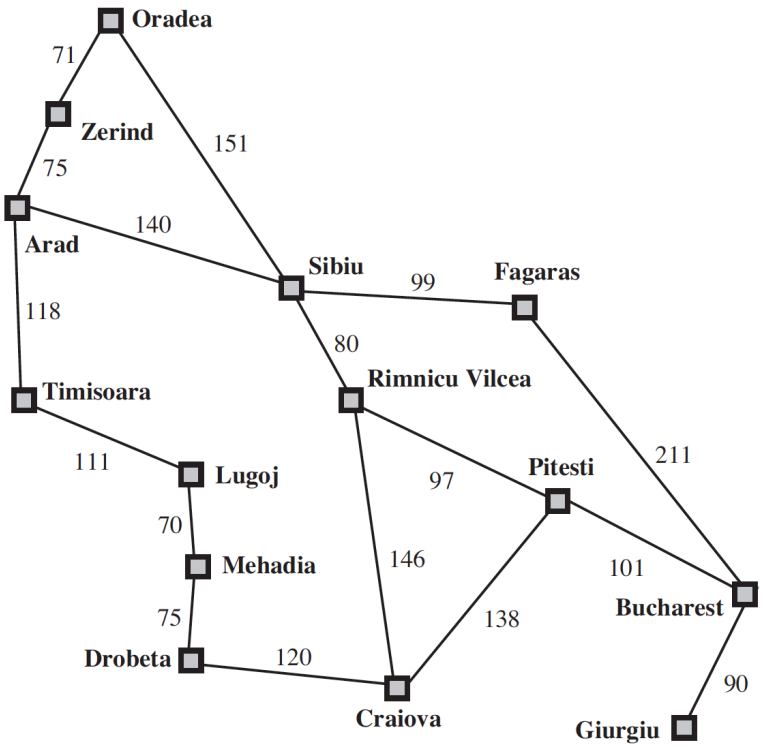
Travelling Salesperson Problem

Find a travel plan that allows visiting every city **exactly once**.



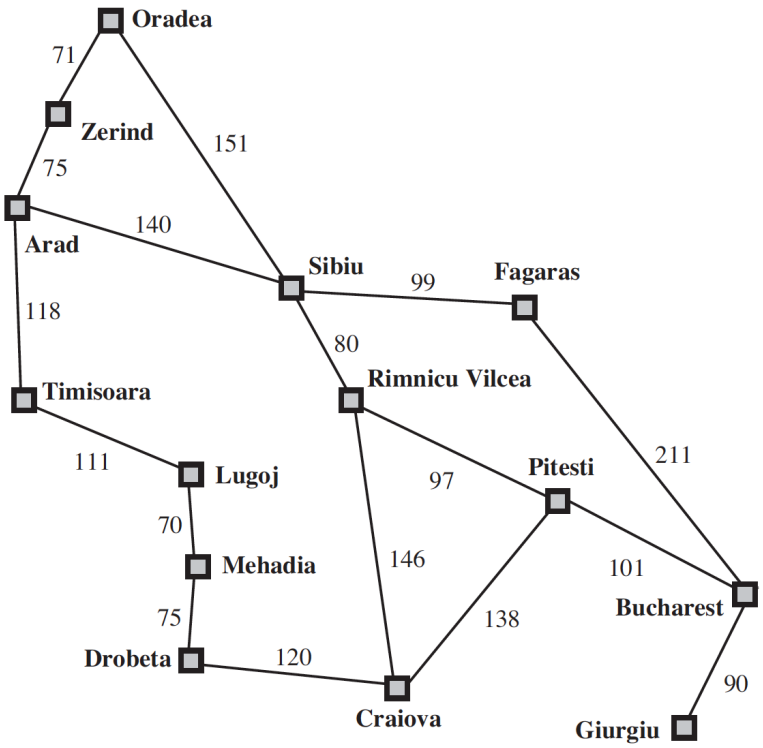
Back to Route-finding Problem: From Arad to Bukarest

Problem Solving as Tree-Search



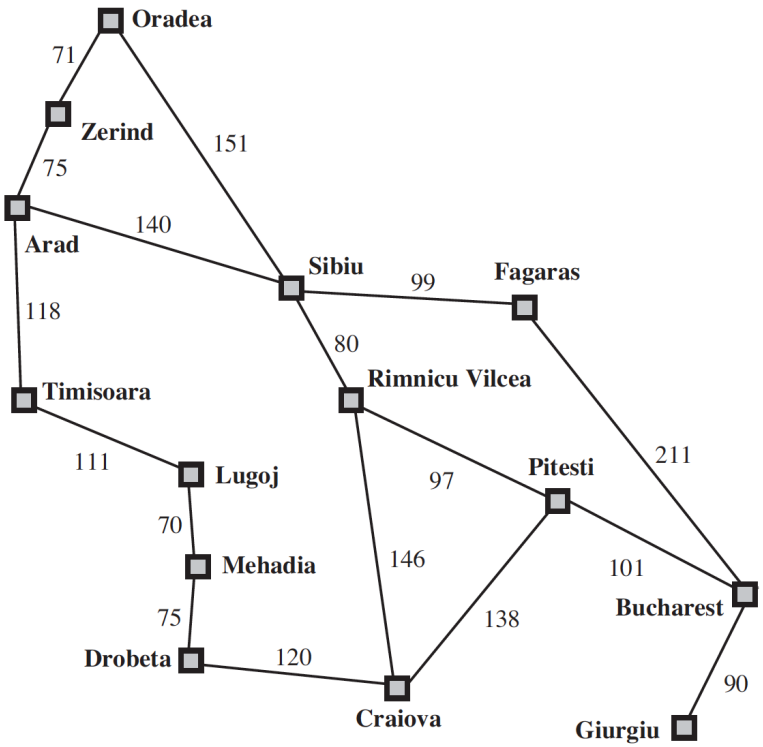
Back to Route-finding Problem: From Arad to Bukarest

Problem Solving as Tree-Search



Back to Route-finding Problem: From Arad to Bukarest

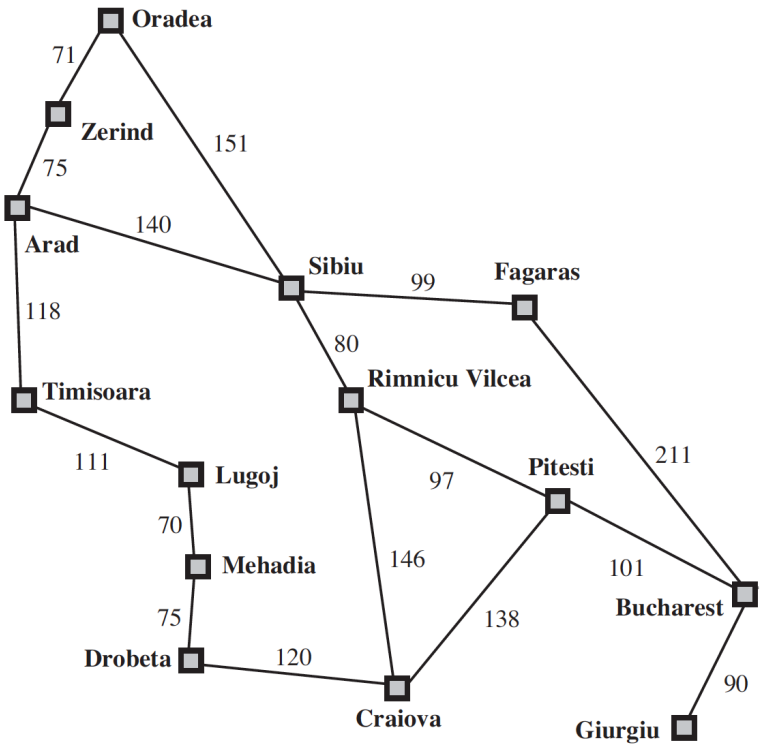
Problem Solving as Tree-Search

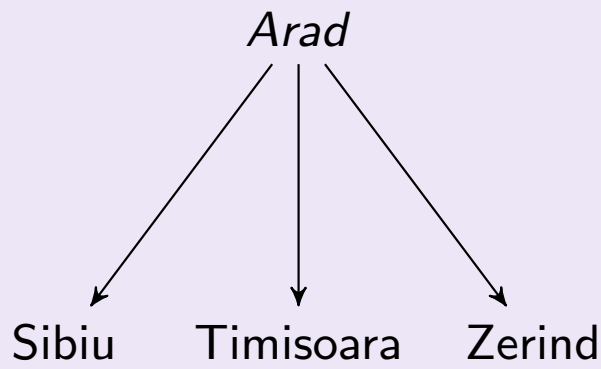


Arad

Back to Route-finding Problem: From Arad to Bukarest

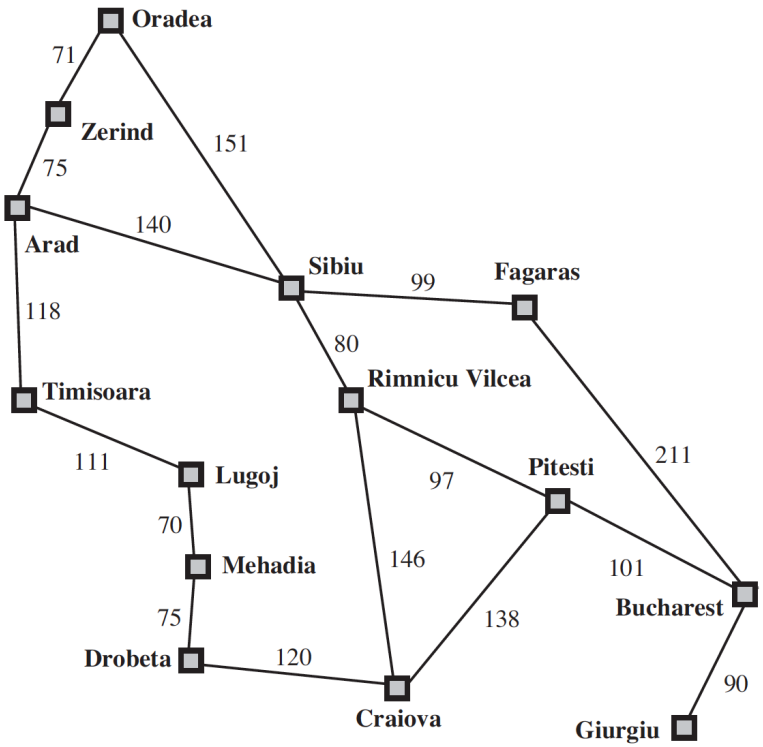
Problem Solving as Tree-Search

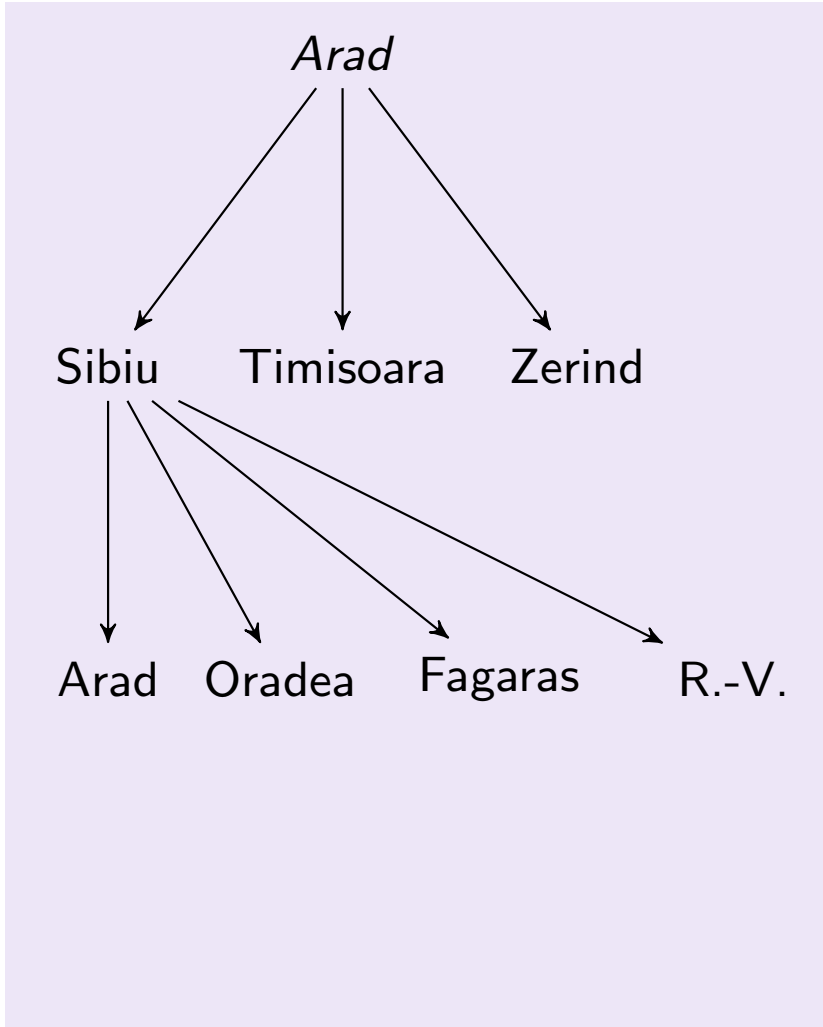




Back to Route-finding Problem: From Arad to Bukarest

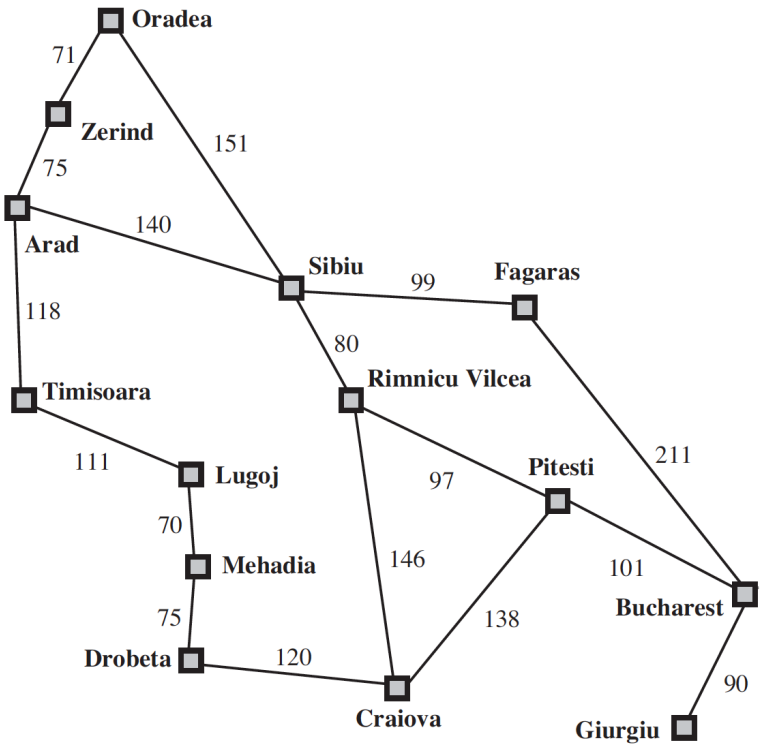
Problem Solving as Tree-Search

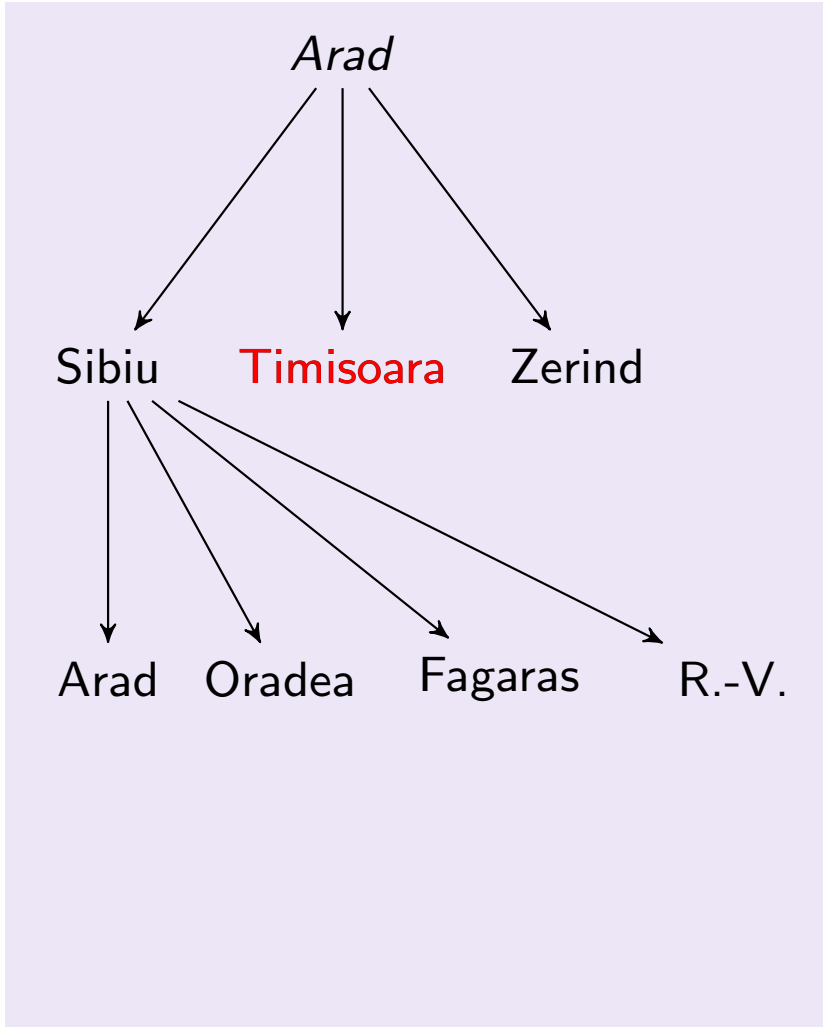




Back to Route-finding Problem: From Arad to Bukarest

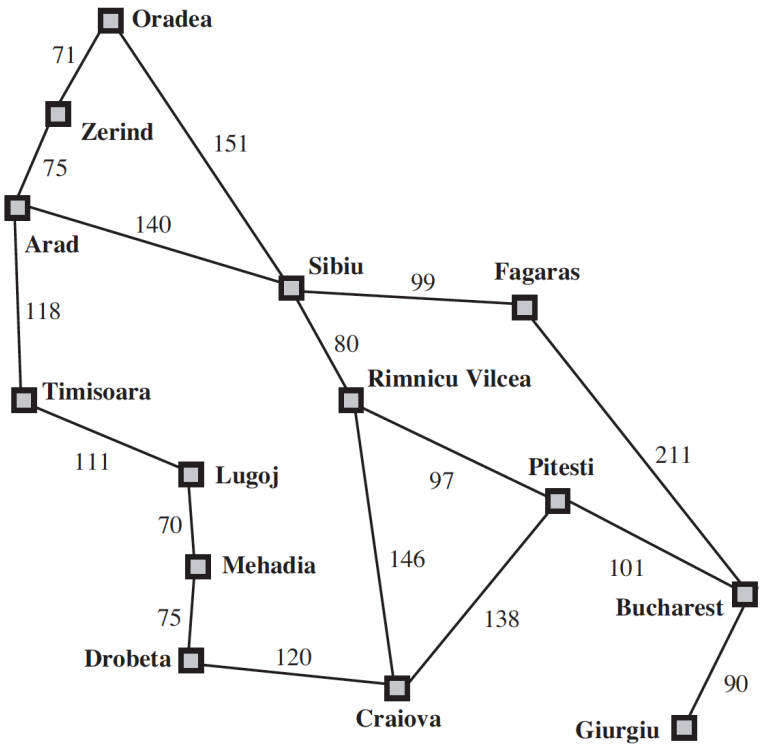
Problem Solving as Tree-Search

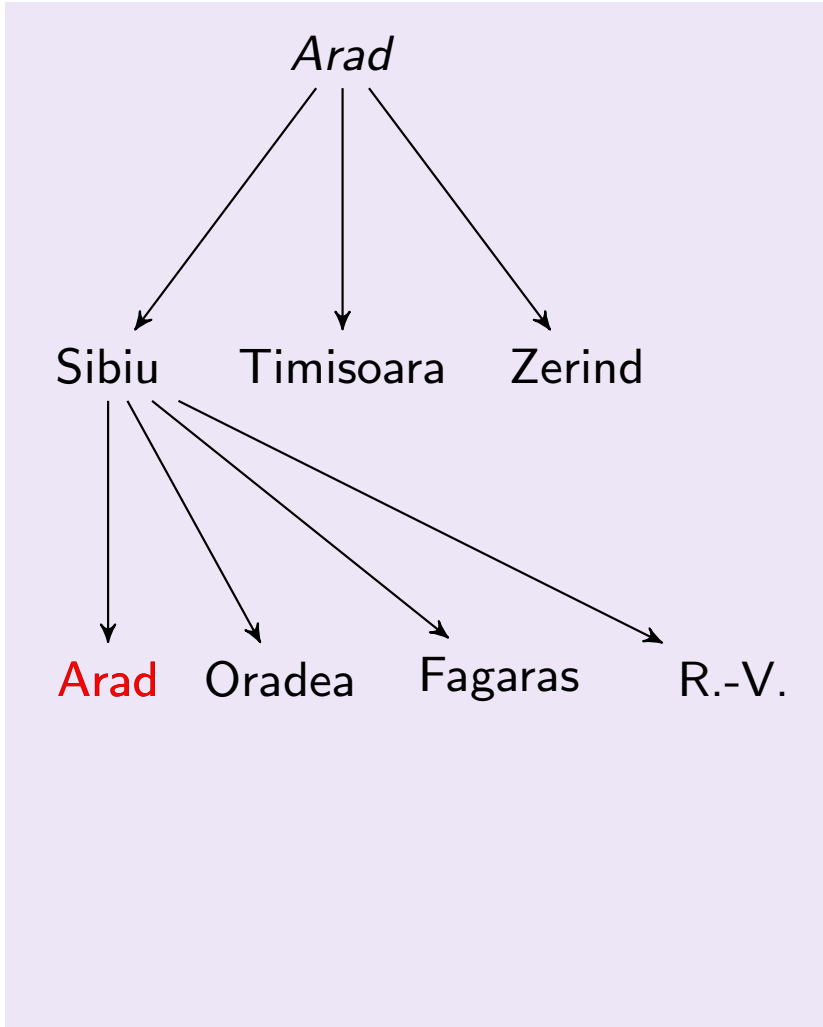




Back to Route-finding Problem: From Arad to Bukarest

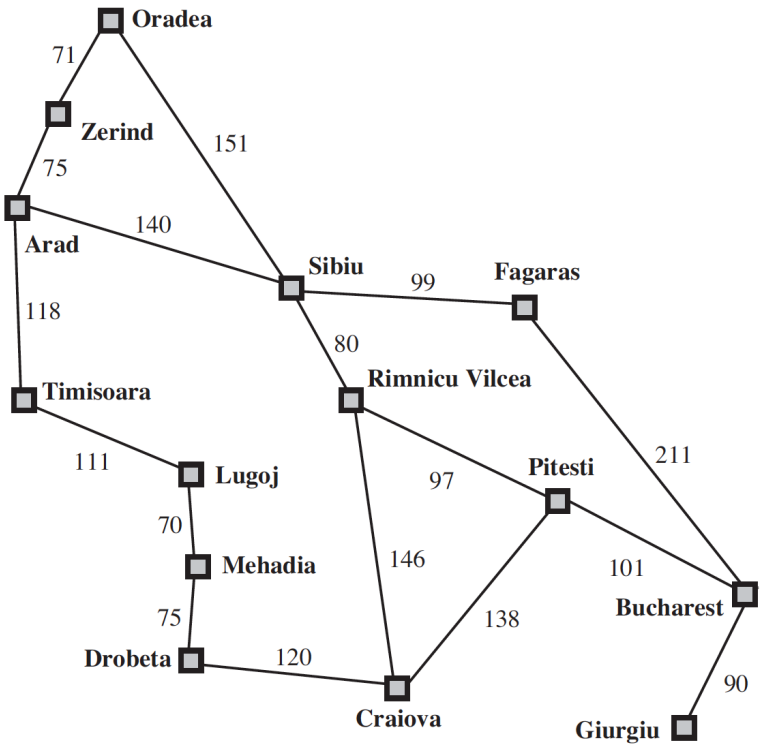
Problem Solving as Tree-Search

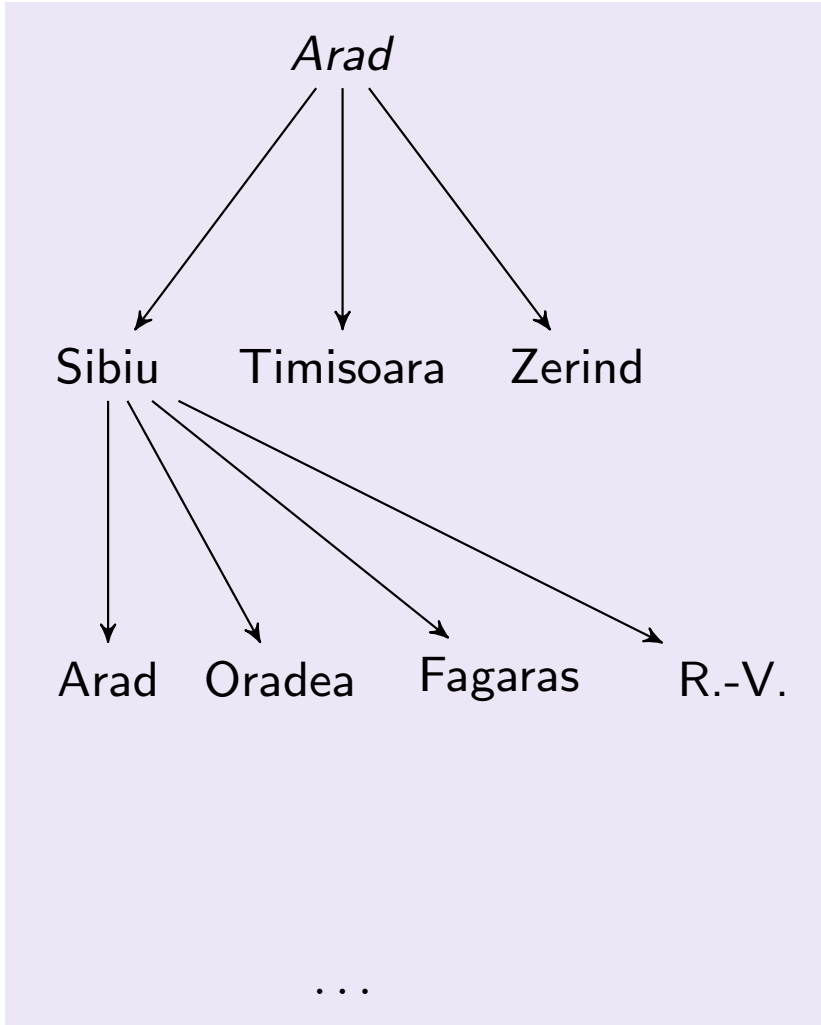




Back to Route-finding Problem: From Arad to Bukarest

Problem Solving as Tree-Search





Expanded nodes & frontier

In tree and graph search, we always generate **all** children of a chosen state s:
We compute $\textsc{Result}(s, a)$ for *all* applicable actions a.
We call this process **expanding** s.

In tree search states are called **nodes**: In tree search we don't keep track of repeated states, and hence distinct tree nodes might represent the same state.

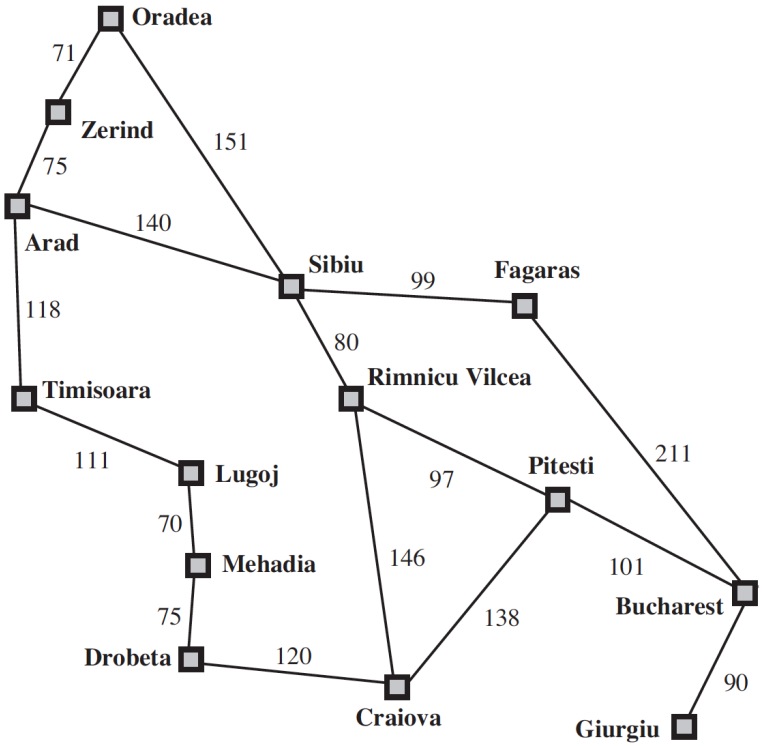
The search has two types of nodes:

- **Expanded nodes**: Nodes for which all children have been generated.
- **Frontier**: Nodes that we generated, but not expanded.

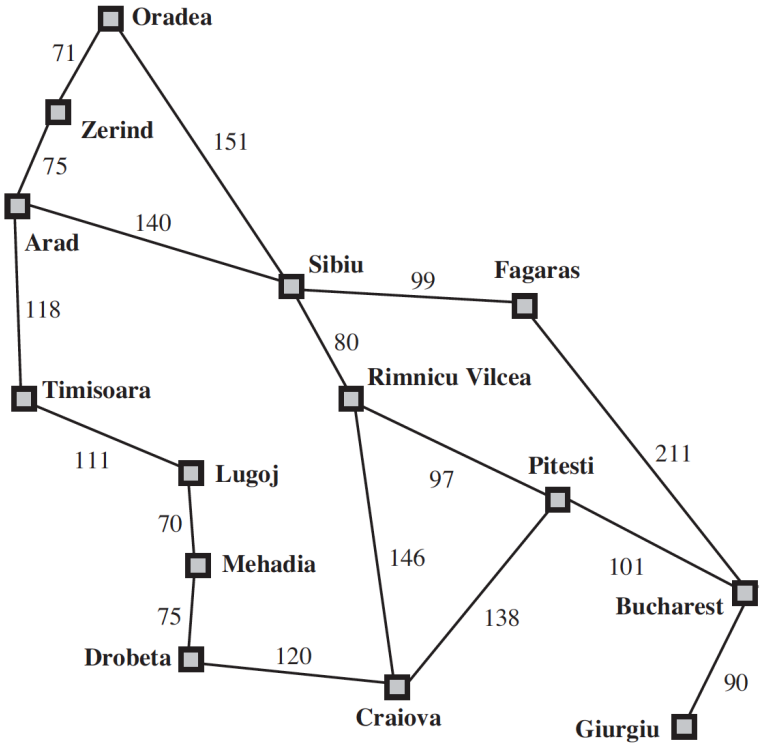
The Tree-Search algorithm

function Tree-Search (problem) returns a solution, or failure
  frontier := $\{s_0\}$ (initial state) // we initialise the frontier
  loop do
    if *frontier* = $\emptyset$ **then return** failure
    choose a node *n* from *frontier*
    remove *n* from *frontier*
    if *n* is a goal state **then return** solution
    for each child *m* of *n* // we expand *n*
      add child m to frontier

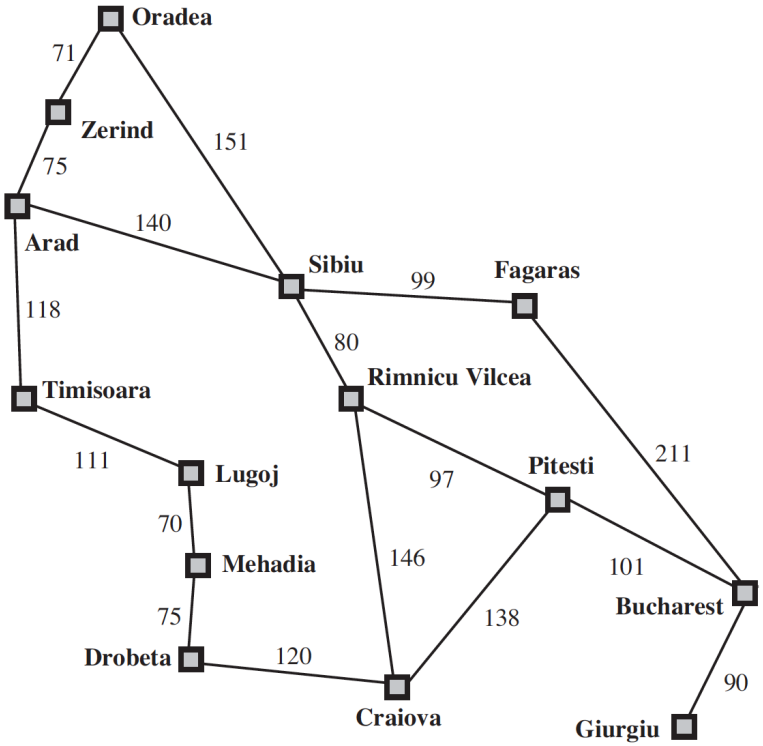
From Tree-search to Graph-search



From Tree-search to Graph-search

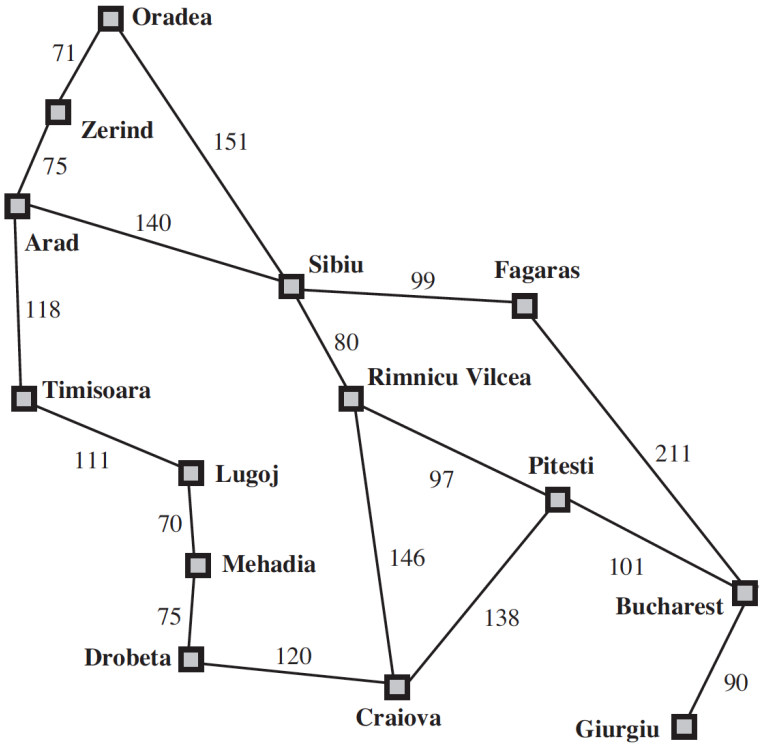


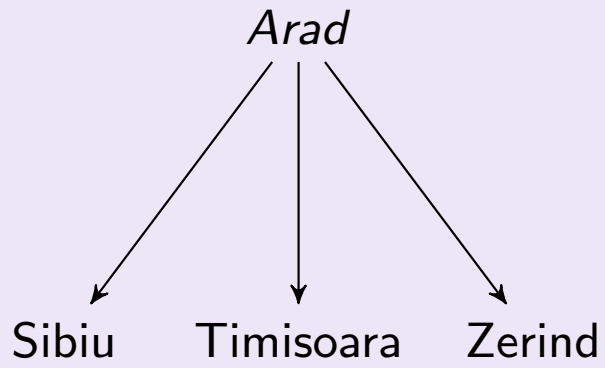
From Tree-search to Graph-search



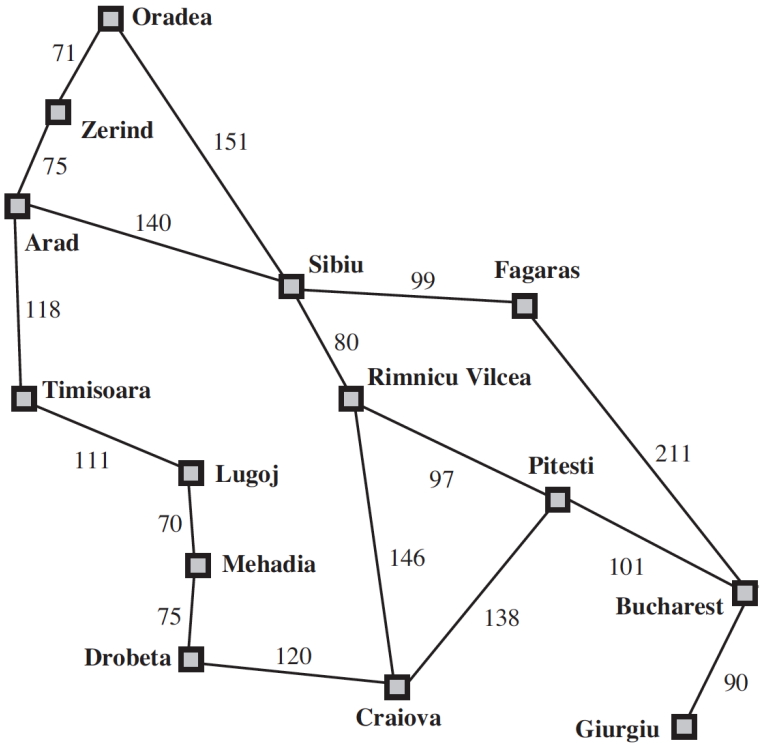
Arad

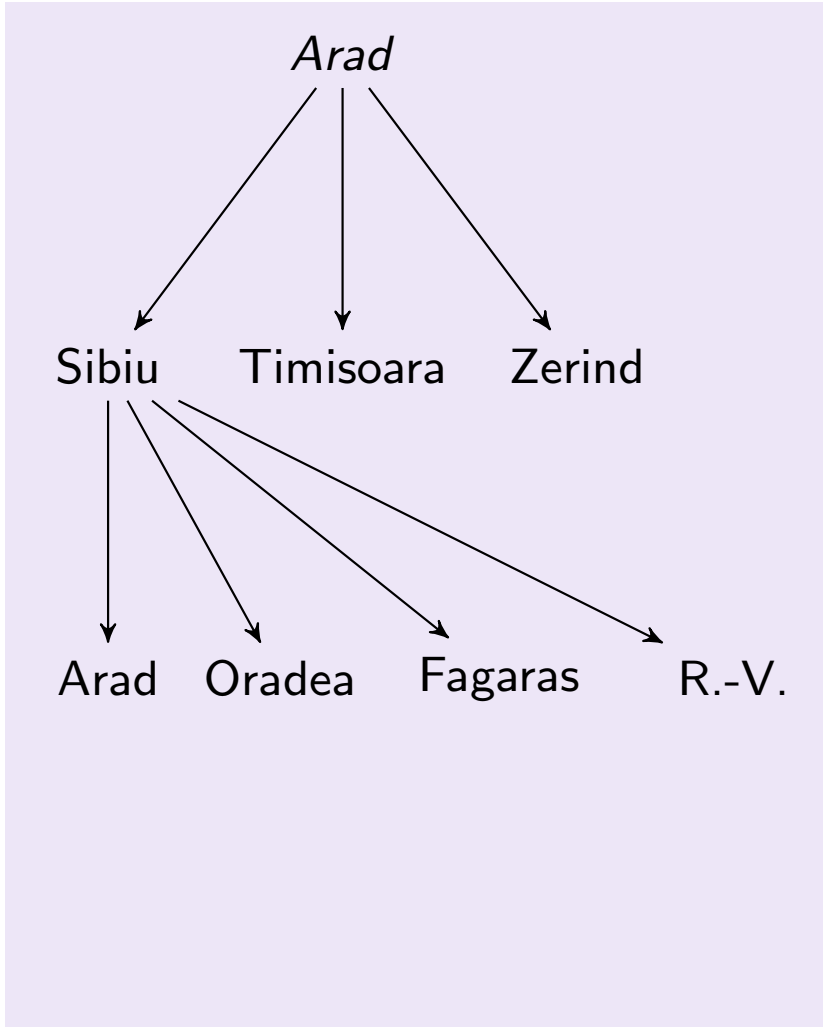
From Tree-search to Graph-search



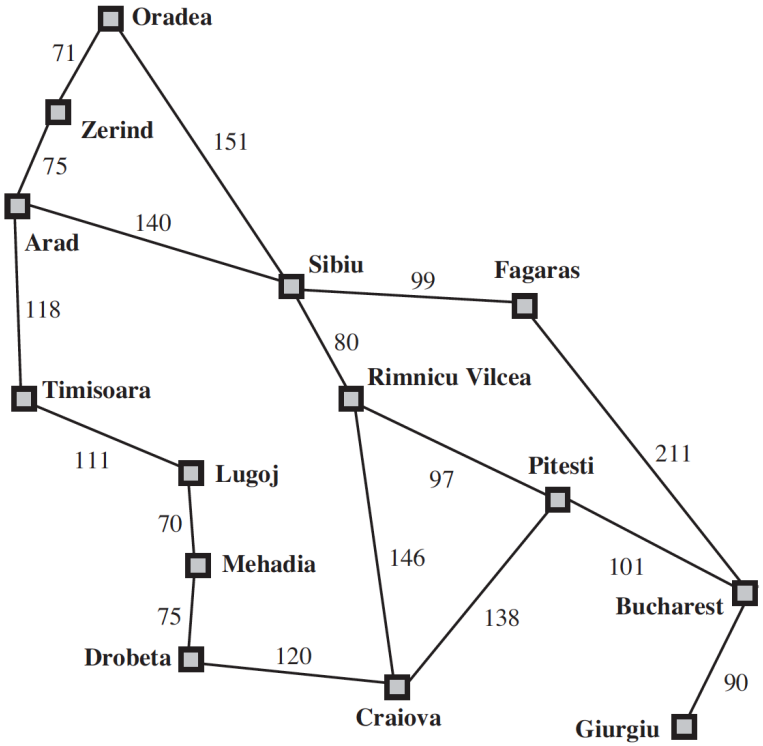


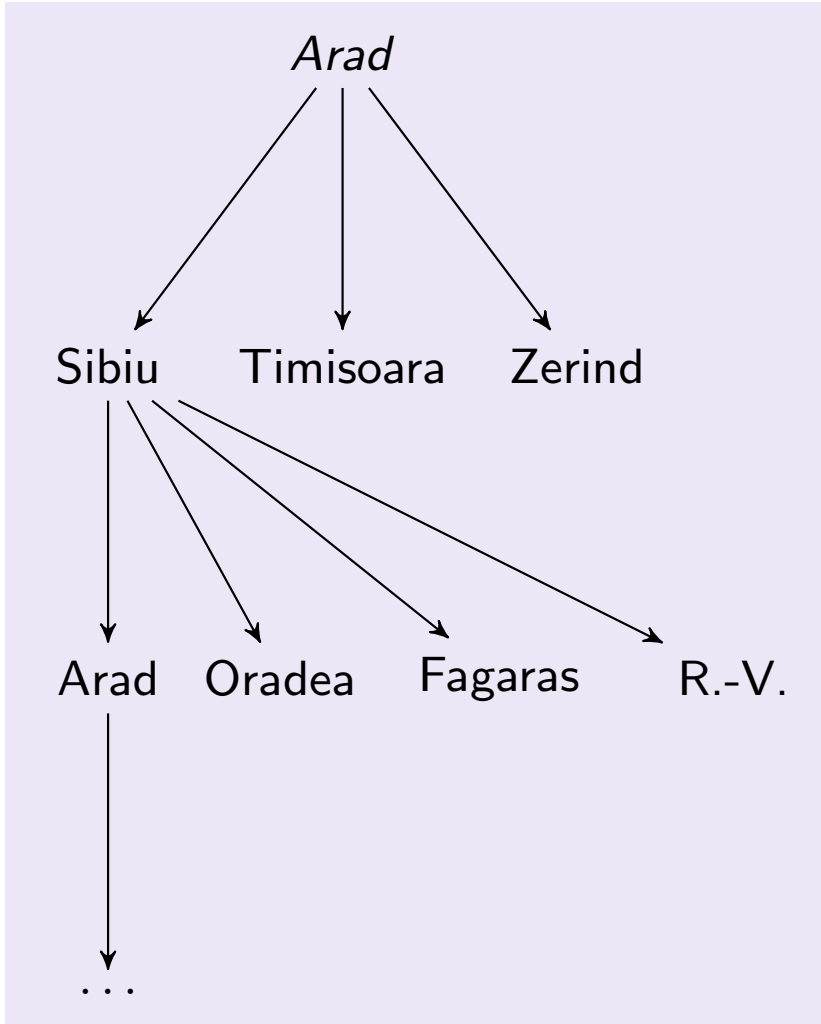
From Tree-search to Graph-search



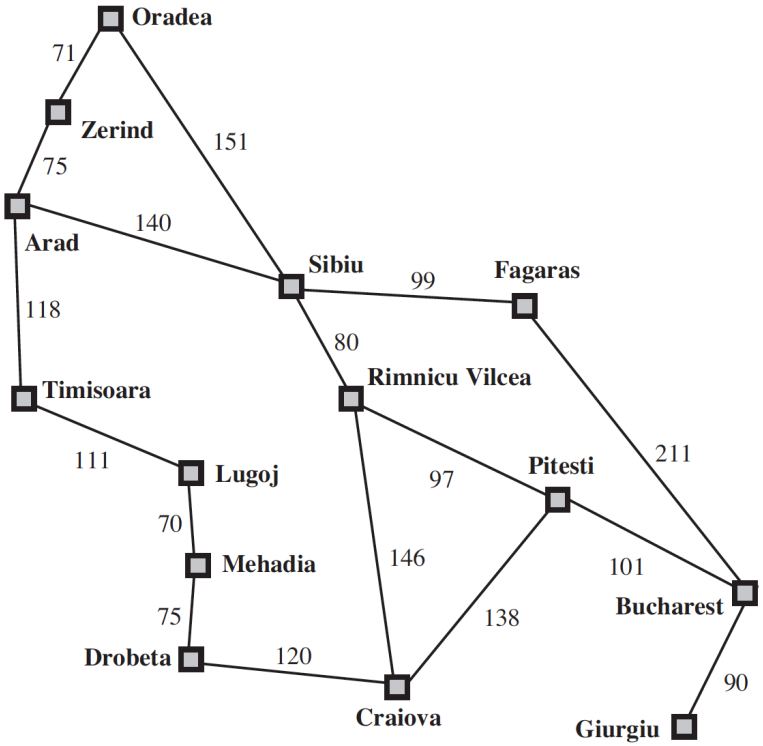


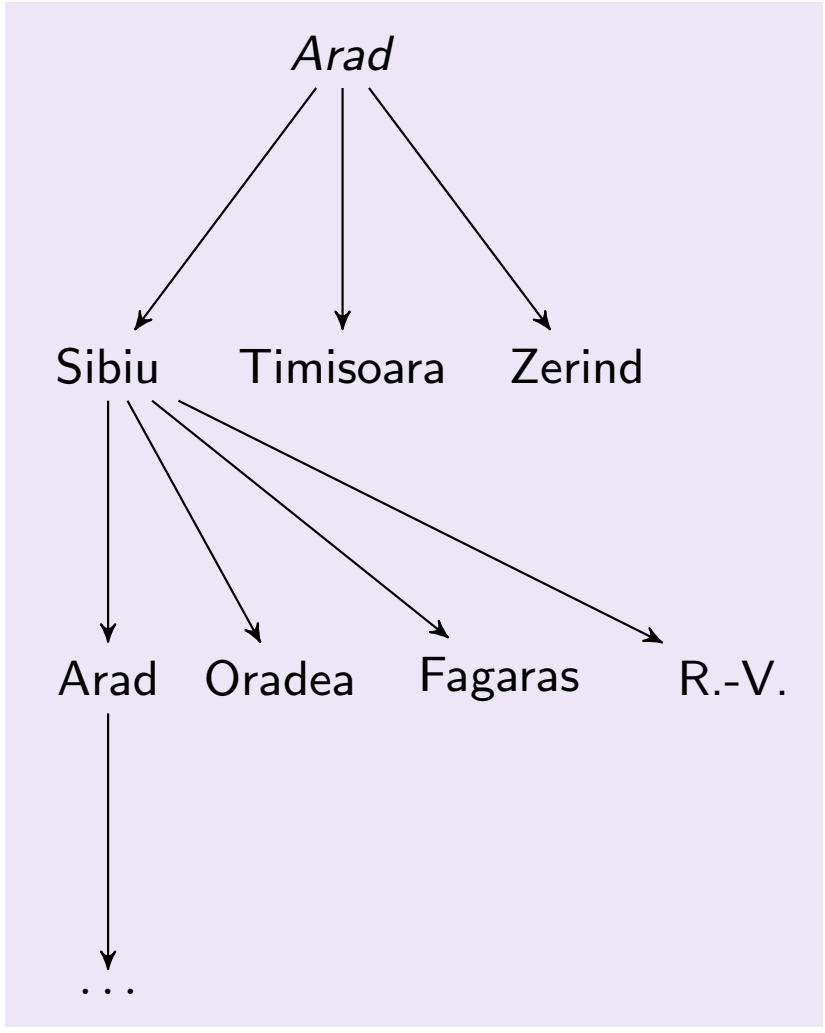
From Tree-search to Graph-search





From Tree-search to Graph-search





Failure to detect repeated states can lead to infinite loops in Tree-Search.
Solution: Keep track of already generated states.
Adding this check to Tree-Search gives us Graph-Search.

The Graph-Search algorithm

The red lines are those added to the Tree-Search algorithm.

function Graph-Search (problem) returns a solution, or failure
  expandedNodes := $\emptyset$
  frontier := $\{s_0\}$ (initial state) // we initialise the frontier
  loop do
    if *frontier* = $\emptyset$ **then return** failure
    choose a node *n* from *frontier*
    remove *n* from *frontier*
    add *n* to *expandedNodes*
    if *n* is a goal state **then return** solution
    for each child *m* of *n* // we expand *n*
      if $m \notin$ *frontier* and $m \notin$ *expandedNodes* then
        add child m to frontier

Solving search problems

Solutions to search problems split into:

- **Tree search**
- **Graph search**

The only difference is: graph search keeps track of repeated states.

Along a different dimension they split into:

- **Uninformed search**: No sense of closeness to the goal.
- **Informed search**: Heuristics used to give a sense of closeness to the goal (e.g. straight-line distance to goal in case of route finding on a map).

For now, we only consider uninformed search.

Along yet another dimension:

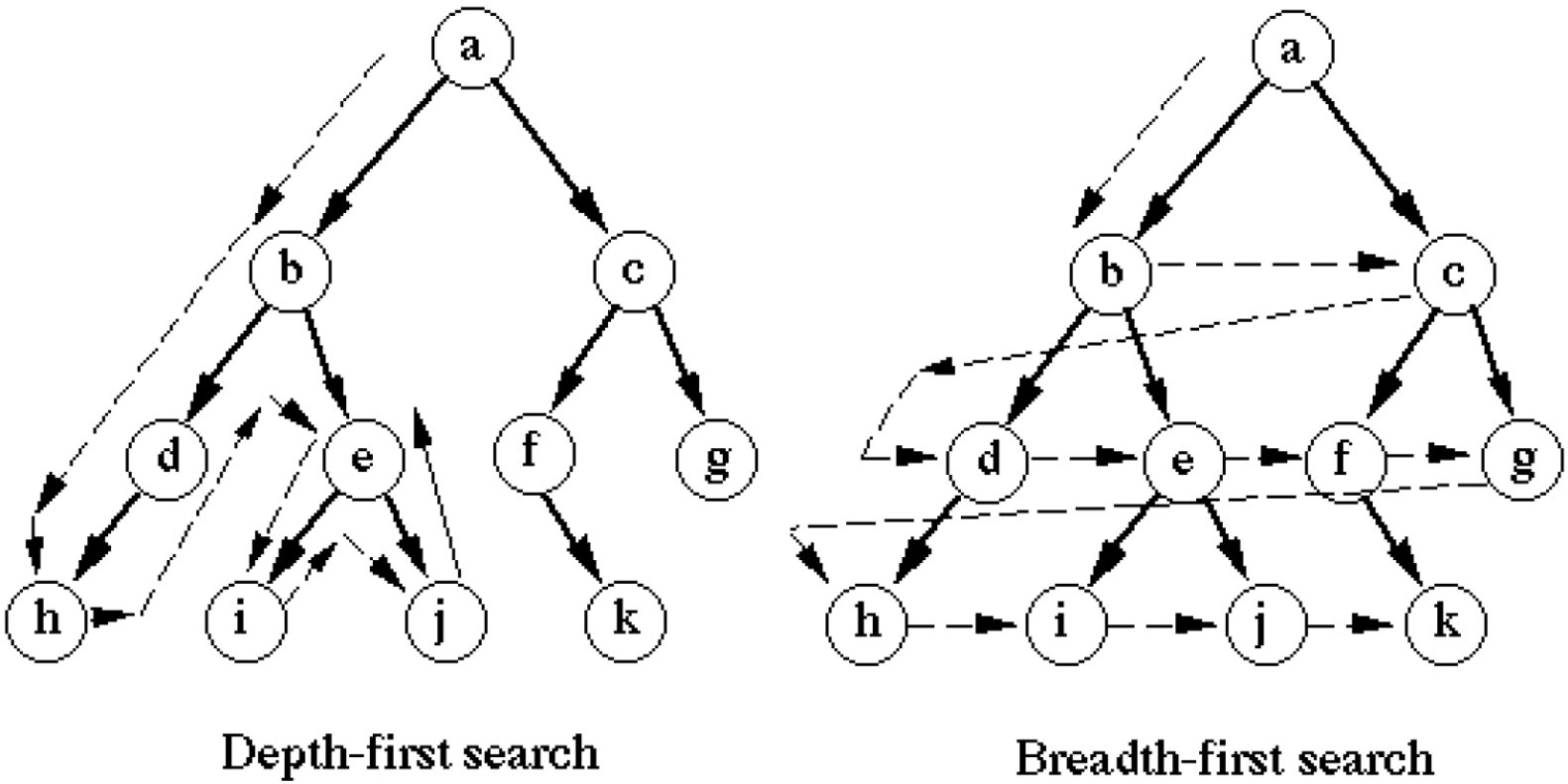
- **Breadth-based search**: Exploring shallow nodes first (close to s_0).
- **Depth-based search**: Exploring as deeply as possible, sacrificing breadth (might miss the goal).

Tree search: DFS vs BFS

DFS: Depth-first search.

BFS: Breadth-first search.

Order of node expansion:



Search strategies of Tree-Search and Graph-Search

Different search strategies can be achieved by simply changing how choose node from frontier and add child to frontier work.

Breadth-first search (BFS):

- Frontier is **queue** (FIFO).
- choose node from frontier: dequeue node from frontier.
- add child to frontier: enqueue node to frontier.

Depth-first search (DFS):

- Frontier is **stack** (LIFO).
- choose node from frontier: pop node from frontier.
- add child to frontier: push node to frontier.

Evaluation of Search Algorithms

Which search algorithm would you rather choose: BFS or DFS?

Go to www.menti.com and use the code 8919 0704

Sokoban

Go to:

https://sokoban-game.com/packs/sokogen-990602-levels/

and complete as many levels as possible, starting with Level 1.

Sokogen: BFS and DFS

level	BFS generated	BFS time	BFS sol length
Sokogen lvl 2	1,984	0.04s	23
Sokogen lvl 3	2,961	0.09s	31
Sokogen lvl 14	19,973	0.14s	37
Sokogen lvl 50	165,389	0.84s	69
Sokogen lvl 76	364,089	1.68s	101
Sokogen lvl 78	613,633	2.88s	51

level	DFS generated	DFS time	DFS sol length
Sokogen lvl 2	2,323	0.04s	55
Sokogen lvl 3	2,410	0.04s	87
Sokogen lvl 14	22,370	0.15s	223
Sokogen lvl 50	156,883	0.61s	883
Sokogen lvl 76	354,483	1.32s	303
Sokogen lvl 78	410,592	1.60s	709

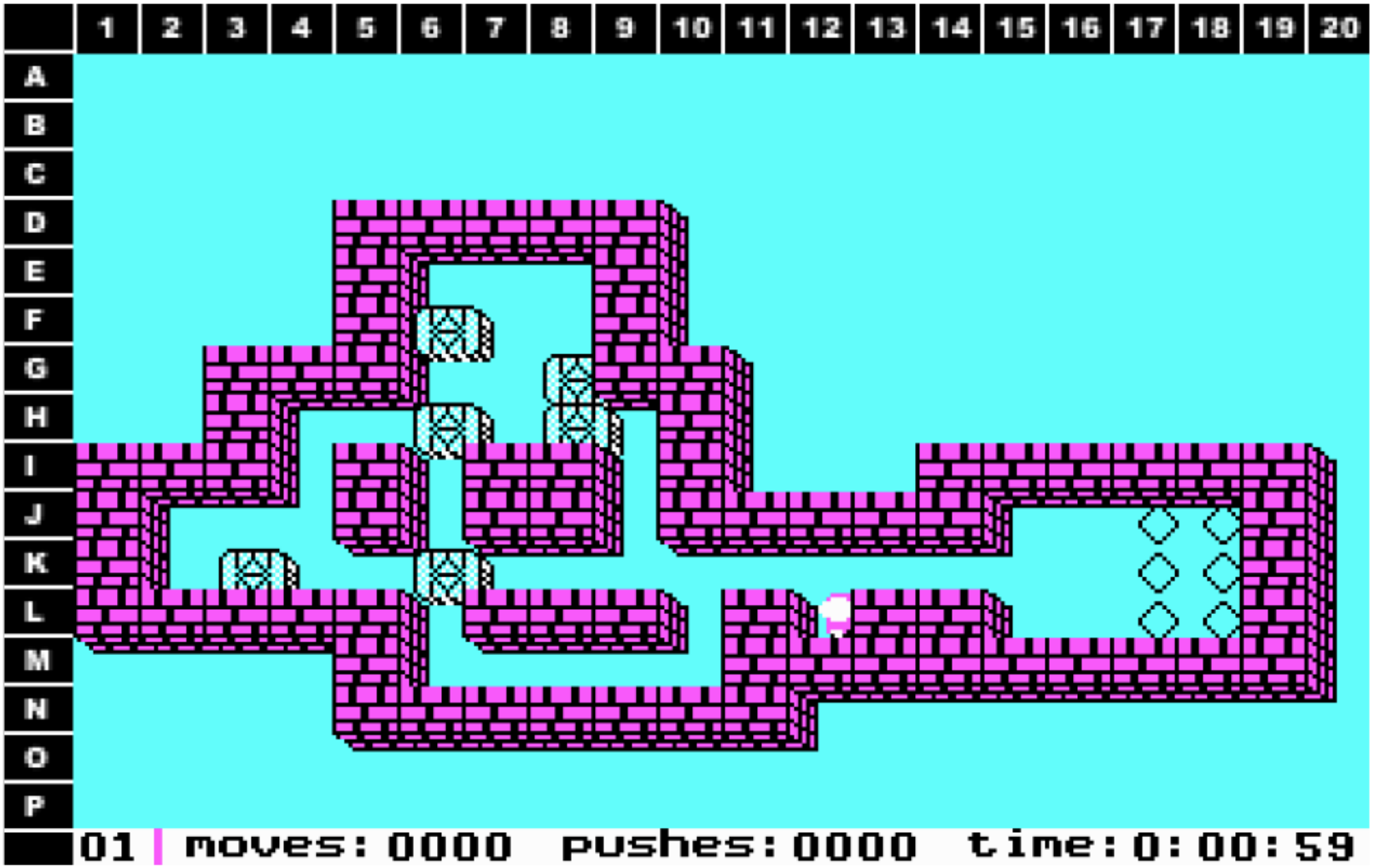
Sokogen: BFS and DFS

level	BFS generated	BFS time	BFS sol length
Sokogen lvl 2	1,984	0.04s	23
Sokogen lvl 3	2,961	0.09s	31
Sokogen lvl 14	19,973	0.14s	37
Sokogen lvl 50	165,389	0.84s	69
Sokogen lvl 76	364,089	1.68s	101
Sokogen lvl 78	613,633	2.88s	51

level	DFS generated	DFS time	DFS sol length
Sokogen lvl 2	2,323	0.04s	55
Sokogen lvl 3	2,410	0.04s	87
Sokogen lvl 14	22,370	0.15s	223
Sokogen lvl 50	156,883	0.61s	883
Sokogen lvl 76	354,483	1.32s	303
Sokogen lvl 78	410,592	1.60s	709

Problems tend to be difficult...

...and so we need better algorithms



Evaluation of Search Algorithms

We can measure problem-solving performance by:

Evaluation of Search Algorithms

We can measure problem-solving performance by:

- **Completeness** Is the algorithm guaranteed to find a solution if one exists?

Evaluation of Search Algorithms

We can measure problem-solving performance by:

- **Completeness** Is the algorithm guaranteed to find a solution if one exists?
- **Optimality** Does the strategy find the optimal solution?

Evaluation of Search Algorithms

We can measure problem-solving performance by:

- **Completeness** Is the algorithm guaranteed to find a solution if one exists?
- **Optimality** Does the strategy find the optimal solution?

  Optimal solution has the lowest path cost among solutions.

Evaluation of Search Algorithms

We can measure problem-solving performance by:

- **Completeness** Is the algorithm guaranteed to find a solution if one exists?
- **Optimality** Does the strategy find the optimal solution?

  Optimal solution has the lowest path cost among solutions.
- **Time complexity** How long does it take to find a solution?

Evaluation of Search Algorithms

We can measure problem-solving performance by:

- **Completeness** Is the algorithm guaranteed to find a solution if one exists?
- **Optimality** Does the strategy find the optimal solution?

  Optimal solution has the lowest path cost among solutions.
- **Time complexity** How long does it take to find a solution?
- **Space complexity** How much memory is needed to perform the search?

Evaluation of Search Algorithms

For Tree-Search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

THE END OF LECTURE 2